S4H: A PEER-TO-PEER SEARCH ENGINE WITH EXPLICIT TRUST

Greg Cowan

Submitted in Partial Fullfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Approved by:
Stacy Patterson, Chair
Buster Holzbauer
Sibel Adali

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

March 2019
(For Graduation May 2019)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank my advisor Buster of course, for the many long and helpful discussions we had about my research and navigating the challenges that arised. I would also like to thank Stacy Patterson for sharing some of her knowledge about distributed systems with me. Also, thank you to Sibel Adali for pointing me towards prior research. And to all three of them for agreeing to be on my thesis committee. Finally, I'd like to thank all the researchers who's shoulders I stand on.

ABSTRACT

S4h is a peer-to-peer search engine with explicit trust. By using a complaint-based reputation system in conjunction with the established peer-to-peer building block of a Kademlia-based distributed hash table, the search engine is built with the ability to use indirect knowledge to remove peers that act maliciously, thereby minimizing common attacks against search engines including censorship and promoting. I developed an implementation of the design and ran experiments to validate that the combined behavior works as the reputation system expects.

1. INTRODUCTION

S4h (an abbreviation for "Search", pronounced "S-4-h") is a design for a peer-to-peer search engine specifically intended to index content-addressed peer-to-peer file-sharing networks. It uses a novel mapping of an existing reputation system onto a distributed hash table, and tweaks them to maintain the desirable security properties of a search engine.

1.1 History Implies Peer-to-Peer

As computing has evolved over the last century, different groups have controlled the human experience of computers by owning the scarce resource of that time. When general purpose computer mainframes became popular in the 1960s and 1970s, the IBM corporation led the field of computer hardware, and therefore dominated the sales of the machines. By 1972, the IBM corporation was America's 5th largest company with a revenue of over $8,273,600,000 [1]. IBM's sales power came from control over processors. However, as microprocessors developed throughout the 1970s, the human experience of computing moved

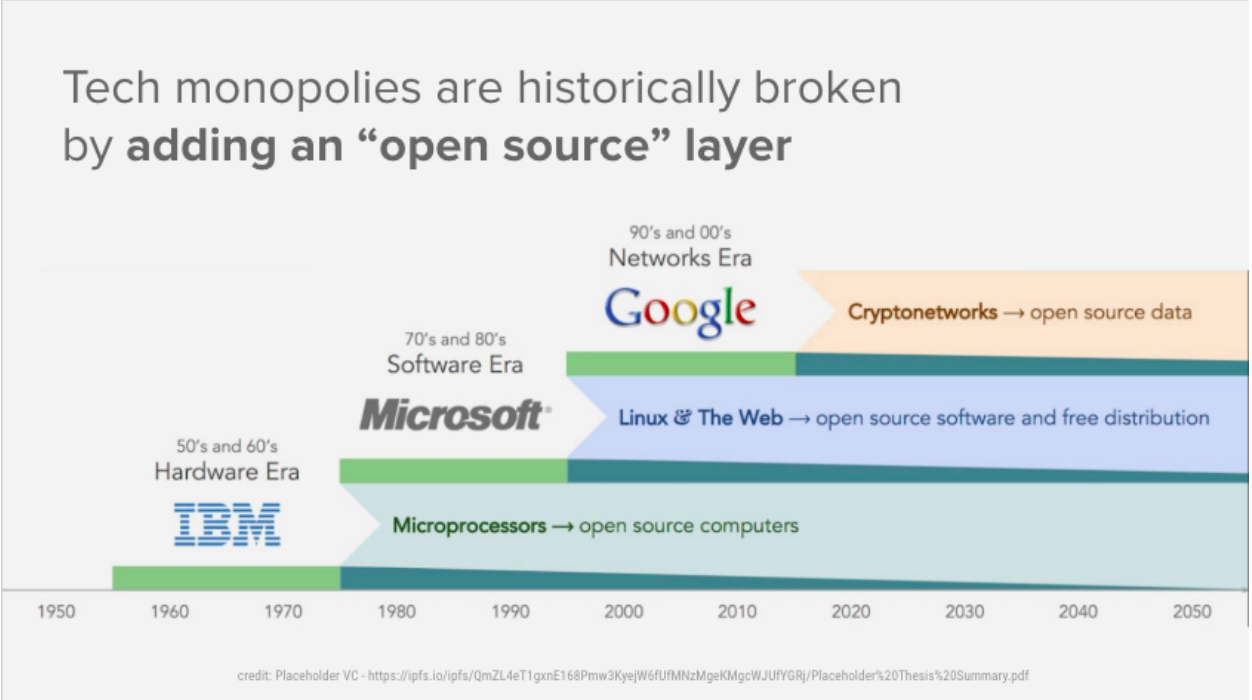

Figure 1.1: Eras of computing monopolies and their commoditization [2].

away from large hardware. By the mid 1980s computing became centered around personal computer software, eventually controlled by Microsoft and Apple throughout the 1990s. As the public-facing Internet developed in the 1990s and 2000s, users' focus shifted from proprietary software to web browsing and mobile smartphones, allowing large companies to develop around services enabled by the control of people's data eventually topped by Google and Facebook by around 2010.

Though IBM, Microsoft, and Apple are all still large successful companies, their historical monopolies on people's access to computing has been broken by "adding an open source layer," as Monegro and Burniske write [2]. Figure 1.1 shows a rough timeline of these companies, their control, and the technologies that succeeded them. Though this is a generalization, and leaves out many important technological developments and companies, it is an illustration of the commoditization of technologies. Just as microprocessors commoditized IBM's mainframes and the Internet and free software commoditized Microsoft and Apple's personal computers, Monegro and Burniske theorize that "cryptonetworks" will commoditize data and therefore eventually undermine Google and Facebook's control [2]. The success of large Internet companies is dependent on the success of their centralized services. Though the Internet started as a decentralized file sharing project, it eventually developed markets for services that were easier to provide with a centralized system. The centralized model is easier to build and monetize, and thereby leads to more development, better products, and more users. This business model has worked well, and continued to scale up, to the point that almost all Internet services are business-owned centralized systems. Culturally, concerns have developed about how much user data these centralized systems collect. Therefore, privacy is currently the main motivation for alternative systems and services. Since the data used in Internet services often originates at the end users, it should be possible for users to provide the information directly to each other, in a decentralized or peer-to-peer system.

1.2 Peer-to-Peer in Real Life

Peer-to-Peer (P2P) systems have not gained much usage except for file-sharing, in which BitTorrent has been a significant source of Internet traffic for the past decade [25]. Though the core technologies for P2P communication are fairly well developed, the user experience is often not as convenient as centralized solutions and the marketing for P2P systems relies on word of mouth more than a large advertising budget. Even given all this,

it is clear that P2P systems can work, because the few success stories like BitTorrent, Tor, P2P multiplayer games, and Bitcoin have managed to break into the public sphere.

While BitTorrent-like file-sharing services have effectively solved the problem of efficiently transferring known files among peers, file discovery is still a mostly unsolved problem. Many BitTorrent client implementations still require a user to lookup a content-ID from a centralized third party indexer. The centralized design of the indexers mean that they can be easy targets for hijacking, censorship, and shutdown. Most famously, the Swedish website The Pirate Bay has been repeatedly shutdown by law enforcement for copyright infringement [26]. Besides those problems, current indexers operate with a informal level of trust. Users must implicitly trust the indexer to provide the contents of the file that they really want, and not the wrong file, or one that has been modified to include malware. Alternatively, users can manually share which indexers to trust, or the website of the original content creator can publish that information. One example of this is a Linux distribution's website publishing their .iso's torrent-link on their downloads page. Of course, P2P systems could simply distribute keyword to content mappings, but the same problems as the current indexers of implicit trust crop up. Therefore, this work addresses the informal question: "Is that file really the one I searched for?" Or more formally, it strives to design a P2P search engine with a trust system.

1.3 Why is Search so Difficult?

When users use traditional search engines like Google Search, Bing, or DuckDuckGo, they must implicitly trust that the search engine is returning results that correspond to the query and are not spam or malware, that the search engine prioritizes proper results over paid advertisements, and that the search engine respects the user's privacy by not selling the user's data. Search engine ranking algorithms are trade secrets, so they have not been audited or peer-reviewed. Even sites like DuckDuckGo, that claim in their privacy policy to not to sell user's data [7], still have to be taken at their word. While it is likely that search engines are striving for the above qualities, it is possible that they are not. With a formal P2P trust system, it is possible to make trust explicit, and block peers that behave poorly from the network.

Theres currently no automated way to answer if a search result is good or not given the search terms. Computer vision algorithms could be used to detect if pictures contain the

search term, or artificial intelligence and natural language processing algorithms could scan for the search query in text documents. Though those approaches could filter out obviously bad results, it would be difficult for them to work well for the subtly bad ones. Pure pattern matching is not the answer either. Two results that both match the query may have vastly different importance. Results may have a cultural importance too, such as a seminal paper in a field of research. But importance may be temporary, national news may be important for a day or for a week, but then gets superseded by newer news. Quality of the source often determines the quality of the search result too. Professional journalistic publications may produce better results than the average blog. Search result preference could be different for different people. One person might prefer political results on one side of the spectrum, while another might prefer the opposite political view. All of these complexities mean that there cannot be correctness defined for search results, like there is for solutions to the problem of content distribution networks, since those networks return results corresponding to a hash value that can be automatically verified. Therefore, search ranking and quality are human or social problems, so our search engines should have human or social components. This work takes the approach of assigning a reputation score to each peer in the network, allowing peers to socially form a network of trust, and exchange search results among those reputable peers.

1.4 Overview of the Design

The solution to the problem of designing a P2P search engine presented here combines a Kademlia Distributed Hash Table [5] for routing and a complaint-based reputation system [6] to form a P2P network that resists common attack patterns. I have tested a basic implementation of the design and shown that it resists these attacks, which is discussed in Chapters 4 and 5.

2. BACKGROUND

2.1 Definitions

These definitions clarify terminology used in this thesis. Some definitions may be more specific here than their colloquial usage.

- Network: computers arranged as nodes in a graph structure, that communicate to other nodes over edges corresponding to Internet links.
- Overlay network: a logical network abstracting over a physical network. Distributed Hash Tables are forms of overlay networks for routing requests. Figure 2.1 shows an example [4].

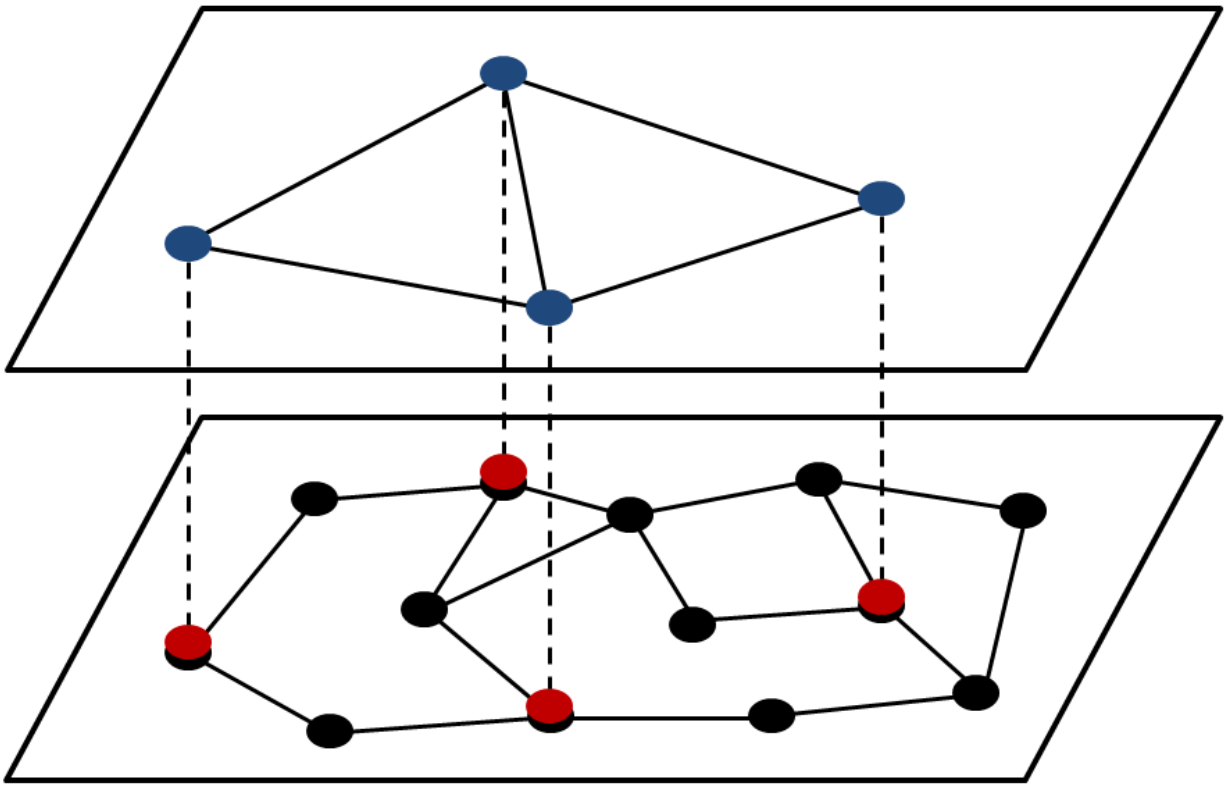

Figure 2.1: Overlay network (shown in blue) abstracting out connections from a physical network (shown in black and red) [4].

- Underlay network: another name for the physical network that an overlay network is built on. The Internet is the largest underlay network, and is built with technologies including TCP, IP, DNS, etc.

- Centralized: a network structure where one or only a few servers service many clients. Also called the client-server model, or the traditional model. This model is the most commonly used one on the Internet. Examples include the RPI CS department's web server, and many other small websites.

- Decentralized: a network structure with many servers and many clients. The servers are usually all owned and managed by a single party. Examples include Content Delivery Networks (CDNs), and Distributed Databases.

- Distributed: similar to decentralized networks, but a distinction between server and client may not be present.

- Federated: a decentralized network in which the servers are owned and maintained by different parties. Examples include DNS, Email, and Mastodon instances [3].

- Peer-to-Peer (P2P): a network in which end users' (as opposed to a service provider's) computers connect directly to one another to cooperatively provide the service. A computer participating in the network is interchangeably called a peer, a node, or a process. One major distinction to decentralized or distributed networks is that since peers are not owned by one party or vetted before joining the network, they can be malicious.

- Malicious node: also known as an adversarial or byzantine node, where the node is breaking the rules of the algorithm, whether to steal information, gain an unfair advantage, attack a specific target, wreak havoc, or for any other reason. Many adversarial nodes may be owned by one malicious operator and coordinate together to perform their attack.

- Churn: the act of peers in P2P networks joining and leaving the network. Churn-rate can describe the amount of nodes joining and leaving during a unit of time.

- Distributed Hash Table (DHT): a common form of P2P overlay network, which routes requests to a target node in $O(log\ N)$ hops, where N is the number of nodes in the network.

- Content distribution service or File-sharing Service: a system that allows the efficient transfer of files among users. Can be centralized (e.g. Dropbox, Google Drive, Cloudflare's CDN) or P2P (e.g. BitTorrent, IPFS, Dat).

- Locality-sensitive hashing: a special type of hash function such that similar inputs produce similar outputs.

- Cryptographic hash function: a function that maps any size input to a fixed size output, and given an output, it is computationally impossible to find the input. It is also computationally impossible to find two inputs that produce the same output.

- Remote Procedure Call (RPC): a network abstraction where a process calls a function on another process over the network, and receives back the return value. Similar to APIs.

- Network Partition: when a network is split into sub-networks because of the failure of a piece of infrastructure.

- Network Address Translation (NAT): when a subnetwork of the Internet is made with its own address space and internal addresses are translated to ports on the external address. This is very common in home or institution routers, but makes P2P connections more difficult, because peers within the subnet do not have a globally accessible address.

2.2 Distributed Hash Tables (DHTs)

Distributed Hash Tables (DHTs) are the cornerstone of modern Peer-to-Peer networks. Despite their name, they do not have nearly as much to do with regular hash tables as they do with network routing schemes. DHTs distribute key-value pairs among peers of a network, and specify an algorithm to lookup values from keys. They are commonly used to build P2P content distribution networks.

The first P2P Content Distribution networks were not fully decentralized (e.g. Napster), or not efficient (e.g. Gnutella), or did not always return results that were in the network (e.g. Freenet).

However, with the release of the four original efficient DHT protocols: CAN [8], Chord [9], Pastry [11], and Tapestry [10], P2P networks gained the tools necessary to build efficient content distribution networks.

A good DHT should have the following properties:

1. **Efficient**: $O(log\ N)$ hops in any route between any two unconnected peers, where N is the number of nodes in the network.

2. **Complete**: returns a value if it exists in the DHT.

3. **Reliable**: tolerant to churn without losing data.

While all of the above four original DHT protocols satisfy these criteria, Kademlia [5] satisfies those too, while simplifying the design. Simplicity makes it easier to understand and implement, thereby making it a popular choice among DHTs [12].

2.2.1 Kademlia

Like all DHTs, peers (or nodes) in Kademlia (Kad) are identified with a node ID, which is a number in the keyspace. The keyspace is from 0 to either 2^{128} or 2^{160}, so that the node IDs are either 128 or 160-bit numbers. Without loss of generality, this thesis uses 128-bit keys. Node IDs are assigned quasi-randomly (see Section 2.2.4), such that nodes are not clustered by geographic location or any other metric. The keyspace can be visualized as a prefix-trie, where the binary string of the node IDs is differentiated by the digit 0 or 1 at each level. Hashtable-like keys also have the same length, and are stored at the peers with the closest node IDs. Distance is defined as the *XOR* of two keys or node IDs, which is easy to compute and symmetric.

Routing is performed by contacting the peers closest to the target key in the current node's routing table. The Kad routing table is designed to have more information about close peers and less about further away ones. It distributes peers into K-Buckets - which are lists of maximum K size - based on distance. Each node stores 128 K-Buckets (though typically not all of them are full) to form their routing table, where other peers are stored at the

K-Bucket corresponding to the *XOR* distance between the current node's node ID and the node ID of the peer to be stored. That means that if the current node's most-significant bit differs from the other peer's node ID's most-significant bit, then the other peer gets stored in the 0th K-Bucket. If the most-significant bits are the same but the second most-significant bits differ the other peer gets stored in the 1st K-Bucket, and so on. In this way, each K-Bucket cuts the remaining keyspace in half. The 0th K-Bucket covers half of the keyspace, the 1st K-Bucket covers a fourth of the keyspace, the 2nd K-Bucket covers an eighth of the keyspace, and so on. Each K-Bucket can hold a maximum of K peers, preferring the oldest still-alive peers. In this way, the Kad routing table stores exponentially less about far away peers, but has a lot of information about close peers [5].

Routing is performed in a series of iterative steps, called a node lookup. At each step, the initiator queries the α nodes that it knows about that are closest to the target key. Those nodes respond to the query with the closest K nodes that they know about to the key. The initiator then repeats the lookup using the new information by querying the closest α un-queried nodes. This algorithm terminates when the initiator does not receive any closer nodes to the target key, in which case it double checks by querying the K closest un-queried nodes, and then terminates [5]. The original paper suggests a K of 20, and an α of 3, but those parameters can be modified depending on the size of the network. This node lookup is called the *find_node* RPC in the Kad paper and is analogous to an iterative DNS lookup. Kad's routing algorithm returns a value if it exists in the DHT in $O(log\ N)$ time, satisfying the correctness and efficient properties.

All RPCs are prefaced by a *find_node* RPC. So, to add a new key value pair to the DHT, a node can do a *find_node(key)*, which will return the K closest nodes to the key to be inserted, and then the initiator node will call the *store(key, value)* RPC at those K closest nodes. RPCs to lookup values and to check if nodes are alive are also defined as *find_value(key)* and *ping* respectively.

To join the network, a peer has to have prior knowledge of a peer's address. This is commonly done by including a hardcoded address in the application for a node hosted by the designer (e.g. IPFS) [13]. This means that the bootstrap nodes end up being very well connected. Then, the new node needs to fill in its routing table, so it should perform a node lookup for its own node ID, thereby filling in peers close to itself. When the *find_node* query for the new node's own ID is received by established peers, they add the new node to their

routing table.

The Kad graph should remain a connected component to maintain DHT properties, since individual nodes only store part of the global knowledge. Therefore, if a node gets disconnected from all of its peers it leaves the network. Also, if a network partition occurs, information stored in the DHT can be inaccessible across the partitions. However, because the node ID selection is independent of geographical location and DHT data is replicated to nearby nodes in the keyspace, a network partition could divide the network such that the components may still have much of the information.

In order to maintain the reliable property, the network needs to be tolerant to churn. That means that data needs to be replicated before all the nodes responsible for it leave the network. To solve this problem, the paper proposes republishing the data every hour [5]. The nodes that store data should republish by using a *store* RPC every hour, unless they receive a republish from another node within the hour. When that *store* happens, peers preface it with a *find_node*, so as to update their routing table. This method allows new nodes to obtain the existing data, while refreshing the K-Buckets. An optimization to reduce the complexity of the refresh *find_node* is published in the Kad paper [5].

2.2.2 R/Kademlia

While the original Kademlia paper only uses iterative routing [5], recursive routing in Kad is also possible as proposed in the R/Kademlia (R/Kad) paper [14]. Recursive routing is especially useful for a well connected permanent node to lookup a value for a transient mobile node. This extension corresponds nicely to recursive DNS queries. The R/Kad paper shows that recursive routing provides lower latency lookups compared to iterative routing. The paper proposes a Direct mode, where nodes along the route relay the closest K nodes that they know about back to the original searcher. That way, the searcher gets as much information as they would using the iterative algorithm, allowing them to populate their K-Buckets. The other method proposed is called the Source-Routing mode, which is more like a traditional recursive query where the nodes along the route forward the closest K nodes that everyone so far knows about and the original searcher only gets back data at the very end. These routing modes are visualized in Figure 2.2. Source-Routing has the benefit that nodes along the route get updated routing information too and that it can better route through NATed nodes [14]. However, the major downside of source-routing that is not addressed in

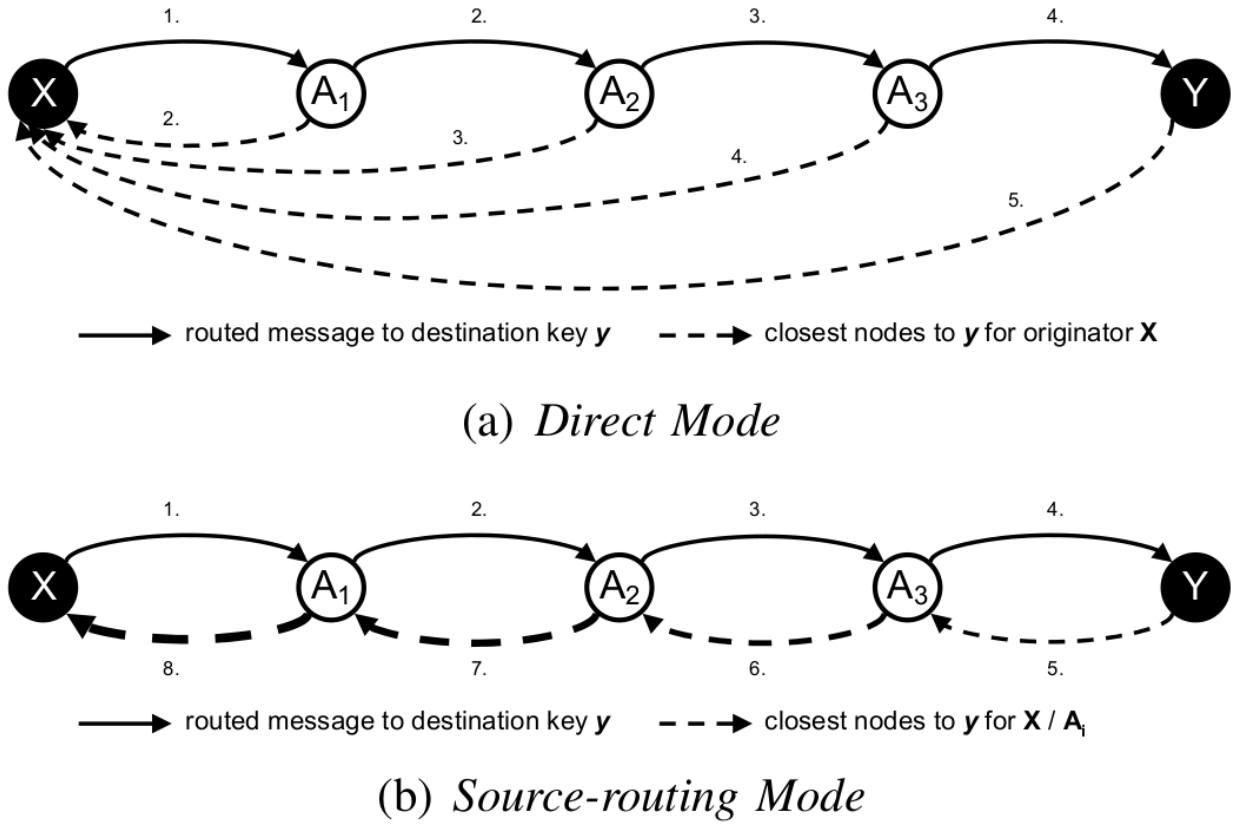

Figure 2.2: Visualizing different modes of recursive routing [14].

the R/Kad paper is that every node along the route has complete control over what routing information to return. An adversary with control over a node in the route could censor the data by replacing the actual K closest nodes with nodes that the adversary controls, thereby gaining control of some of the searches to that key. For that reason, iterative or Direct routing is preferred in this thesis.

2.2.3 Security in Kademlia

Though the Kademlia DHT satisfies the properties: efficient, complete, and reliable as defined in Section 2.2, its design does not prioritize security, and generally assumes that peers follow the algorithm correctly and are not adversarial. Real-world P2P networks are open to use by anyone, so I add another DHT property:

4. Secure: resistant to public and targeted attacks (as defined later in Section 2.2.3).

The original Kad paper leaves *find_node* queries open to hijacking, where an adversary along the route may return K nodes that are very close to the target, which are owned by

the adversary. In this case, the adversary controls the K nodes closest to the search key that the searcher node knows about, and therefore the adversary can arbitrarily return whatever they want. This kind of attack on a query is defined here as *hijacking a query*.

Hijacking a query is possible in Kademlia because nodes can choose their own node IDs, and creating new peers is trivial. An adversary could create many peers around target keys, in order to hijack the queries for them. Kad does come with a mitigation, which is that when updating the K-Buckets in the routing table, a node prefers its oldest still-alive peers over new peers. Consider if a node's K-Bucket is full of K non-adversarial peers, and a new peer (which could be adversarial) is situated to be added to that bucket. By the Kad algorithm, it will be added if the K-Bucket is not full, but if it is full, then the node will ping the least recently contacted peer in that K-Bucket. If the least recently contacted peer responds (which it will if it is still up), then that least recently contacted peer is kept, and the new (potentially adversarial) peer is not added. Otherwise, if the least recently contacted peer does not respond to the ping, then it is dropped from the K-Bucket, and the new peer is added in its place [5]. This preference for oldest still-alive peers means that an adversary cannot instantly take over an established Kad network by flooding it with new adversarial nodes. However, once the adversarial nodes get added to some routing tables, the original Kad paper does not provide a way to prevent them from hijacking a query, report their bad behavior, or prevent them from performing other attacks. This attack of creating many adversarial nodes is called a *Sybil Attack* [16], and choosing node IDs close to a target is called an *Eclipse Attack* [15].

Attackers may be motivated for many different reasons, and are amazingly creative with their attacks. The following is a non-exhaustive list of attack types that are applicable to P2P networks and search engines:

1. **Denial of Service (DoS)** attacks are when the attacker prevents the normal operation of the system. It may be targeted against one peer, affect the whole network, or somewhere in between.

2. **Censorship** occurs when an adversary prevents access to specific information. It could be cultural censorship of taboo media, political censorship of the opponents, or corporate censorship of competitors. Censorship targets specific subjects and search terms, or targets specific peers that are originators of that data.

3. **Promoting** is the inverse of censorship. Its goal is to make the target information highly visible. Examples include excessive political propaganda or corporate advertising.

4. **Amplification** is a kind of attack can be performed on any server-like networking process that replies to requests. In this case, an attacker would forge IP source addresses on small packets to peers, so that the peers respond with large responses to the target IP. This kind of attack is often used to increase the size of DoS attacks.

5. **Data Mining** is when an adversary steals personal data of a user or users. This kind of attack is applicable to search engines, often in order to enable targeted advertising.

Many of these attack patterns are more effective with more adversarial nodes in the network, especially censorship, promoting, and data mining. Therefore, the motivation for resistance to Sybil and Eclipse attacks is established.

Other attacks on the underlay network that the Kad network is built on are possible too. One common attack tool when using IP as the network layer is IP address spoofing, which is when the host sending network packets says that the source is a different IP address than its actual IP address. This makes it seem like the packets are coming from a different host. IP address spoofing can be used for Amplification attacks or man-in-the-middle (MITM) attacks, in which a host in the middle of the connection eavesdrops and tampers with the data being sent between two communicating hosts. Replay attacks are used against authenticated and encrypted packets. If a host signs (and potentially encrypts) some data and sends it across the network, a MITM may send that authenticated data multiple times, or to a different target than intended. However, the fix for this kind of attack is straightforward. By including random data - called a nonce - with the real data, a receiving host can throw out packets that have duplicate nonces. Also, signing the packet header, including the source and destination IP addresses and ports can prevent a MITM from forwarding authenticated data to the wrong host.

2.2.4 S/Kademlia

The S/Kademlia (S/Kad) paper extends Kad with some security focused modifications [15]. S/Kad notes that the network layer needs to provide authentication, encryption, or prevent IP address forging, and notes that an insecure underlay network can be used to attack

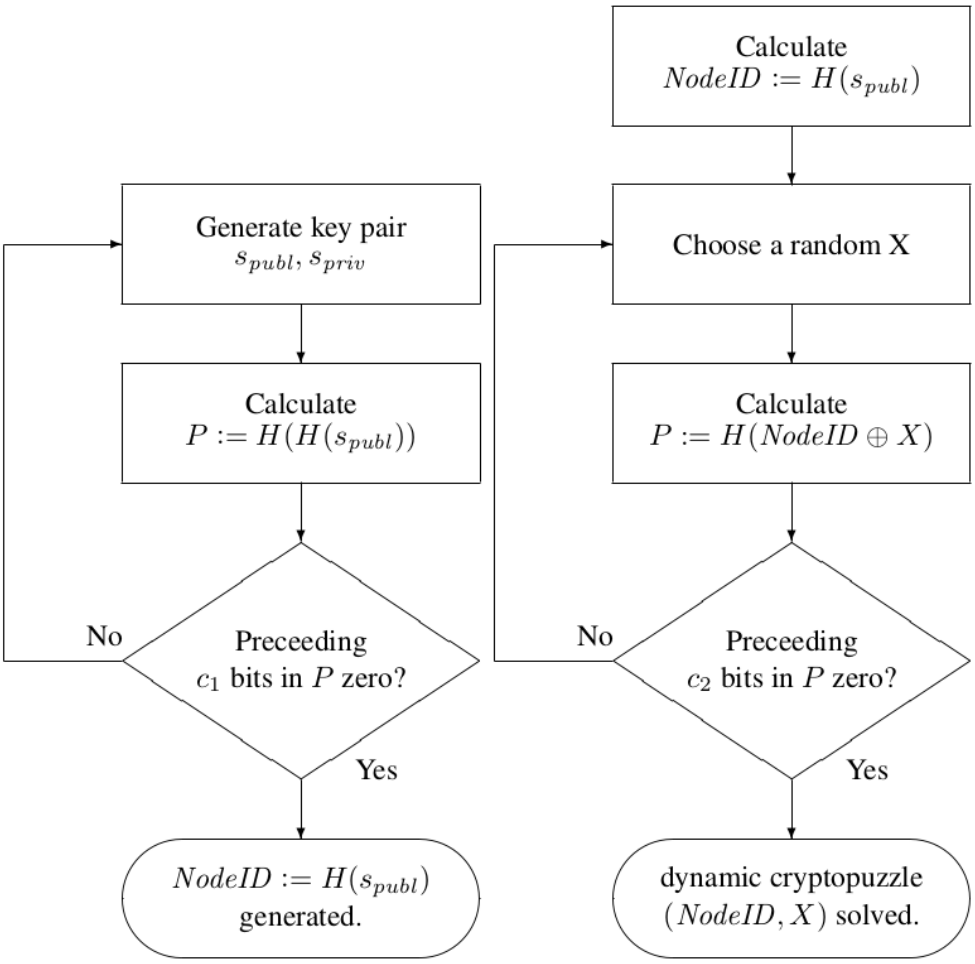

Figure 2.3: Static (left) and dynamic (right) cryptographic proofs-of-work algorithms for securing node ID generation [15].

a Kad overlay network unless proper mitigations have been used. S/Kad first addresses the problem that nodes can pretend to be other nodes, i.e. there is no authentication. Therefore, it proposes that each node use public key cryptography to sign messages, including the IP address, port, timestamp, and message contents, to prevent address forging, replay attacks, and man-in-the-middle attacks. Then, resistance to Sybil and Eclipse attacks is developed by making it difficult to generate many nodes, and impossible to choose node IDs respectively. Sybil and Eclipse attacks can be prevented in two ways, by either getting a centralized certificate authority (CA) to sign the public key, or by using cryptographic proofs-of-work (called crypto puzzles in S/Kad) to generate node IDs. S/Kad proposes using the certificate authority method at the beginning of the network, when it is small [15]. The centralized certificate authority design does not scale well with the size of the network though, and

comes with the discussed downsides of a centralized model. It is also unclear how a certificate authority would prevent many nodes being generated by the same adversary. The CA would check IP addresses with a round trip handshake, but it cannot force only one node per IP address, because of the prevalence of NATs.

Cryptographic puzzles work by mathematically forcing node ID generation to be highly improbable, so that it takes many tries to successfully generate a node ID. This technique has the benefit of being local, so it works well for distributed networks. S/Kad proposes two crypto puzzles, a static one to make it hard to choose node IDs near a target, and a dynamic one to make it hard to generate many node IDs. Figure 2.3 shows these two algorithms, where H is a cryptographic hash function, $\oplus$ is the XOR operation, and c_1 and c_2 are difficulty parameters. Running the puzzles take $O(2^{c_1} + 2^{c_2})$ time complexity. Then, when a peer is being added, it sends its public key, node ID, and random number X to the current node, and the current node can verify that it used the puzzles in $O(1)$ time [15]. These cryptographic proofs-of-work do not perfectly prevent Sybil attacks though. A sufficiently powerful adversary could still generate many node IDs, even though it is computationally expensive. The difficulty of solving the puzzles could be tuned very high by increasing c_1 and c_2, but then regular users would not have the computational power to generate a node ID. However, if the difficulty is low enough, then a powerful adversary could still generate many node IDs.

S/Kad brings up that in Kad's *find_node* lookup, once an adversary is queried along the path, it can return K of its adversarial peers close to the target key, thereby hijacking the query [15]. Though the authentication and crypto puzzles make it more difficult for adversaries to generate those peers, it is still possible. Therefore, S/Kad proposes to split the query into d disjoint paths. In this algorithm, the initial K peers from the searcher's routing table are split into d groups, and then d regular Kad lookups are performed with those initial groups, except that peers can only be used in one of these queries, so that the lookups are distinct [15]. Distinct queries mean that if one gets compromised by an adversary, the other queries are not necessarily compromised. Of course, it is not always possible to detect when a peer is behaving poorly (because they are an adversary), so this presents the problem of merging the results at the end of the queries.

2.3 File-sharing Networks

File-sharing networks were the inspiration for this thesis. Existing systems like BitTorrent [12], IPFS [13], and Dat [17] work well for sharing files, given the content ID of the file. However, none of them currently provide compelling solutions to get content IDs from searches. Standard BitTorrent user interaction starts with a user obtaining a torrent file (which contains the content ID and some metadata) from a centralized third-party tracker. The Tribler BitTorrent client provides a search feature based on a P2P gossip protocol, but does not currently have mitigations for many of the possible attacks on such a system without a trust model [18]. The Dat Project targets the sharing of scientific data use case, so the documentation suggests manual sharing, such as sending a content ID to your friend over a chat application, email, or by publishing it on a website [17]. The IPFS documentation suggests the same thing, just sharing content IDs through some other means [13]. In all of these systems finding the content ID of the file that a user is looking for is difficult. This difficulty provides the motivation for this work.

2.3.1 More on Identifiers

Content IDs are simply strings that uniquely represent some data. Often these strings are 128-bit numbers, because they are calculated by running a cryptographic hash function on the contents of a file. Cryptographic hash functions such as BLAKE2 [20] and SHA-3 [19] map any size input data to a fixed size output number. Secure cryptographic hash functions like those two do not have any known collisions between outputs given different inputs. They also do not preserve locality, such that if one bit of the input is different, two drastically different outputs are returned. Locality-preserving hash functions exist (e.g. TLSH [21]), but are not as secure because they produce similar outputs given similar inputs. Therefore, cryptographic hash functions are effectively one-way functions, in which the output is easy to compute given the input, but the input is computationally impossible to compute given the output. These properties make them great for uniquely identifying files. A file can always be represented as a binary string, corresponding to the hash of its contents as they are stored on digital devices. Therefore, when that content string is run through a cryptographic hash function, a short unique content identifier (ID) is produced. To distribute the load of serving large files among peers, most file-sharing networks split the file into chunks, where each chunk gets its own content ID [13]. Then, the content ID of the whole file can be the hash of a

file that lists all the chunk's content IDs. Versioned files can also be built using this system (e.g. IPFS commit objects). Directories can be encoded too, as a tree of content IDs (e.g. IPFS tree objects) [13].

Content IDs can be compared to traditional file identifiers on the web, called location IDs here. Location IDs identify a file by the location from which it can be obtained. For example, the location ID "https://www.rfc-editor.org/rfc/rfc793.txt" identifies a file *at* that domain and path, regardless of the contents of that file. Of course, we expect that file to actually contain the textual data of IETF RFC 793, but that is not what the location ID asks for. Location IDs are analogous to asking the question: "What is the file at location X?" Content IDs are analogous to asking the question: "What is the file whose contents hash to X?"

Another form of identifier is called a signature ID. A signature ID is the hash of the public key of the peer that is publishing the file. If the public key is verified to belong to a user, and the file is signed by the private key corresponding to the public key, then the file can be identified by the hash of the public key. Signature IDs are analogous to asking the question: "What is the file published by X?" or more specifically: "What is the file that can be verified by a public key that hashes to X?"

Traditional location IDs specify where a file is, regardless of who is serving it, or what it is. Content IDs specify what a file is, regardless of where it is stored, or who is serving it. Signature IDs specify who published the file, but not where it is stored, or what it is, or who is serving it. Using location IDs makes sense for the traditional centralized web, because historically one server would host each file, and clients would ask that server directly for a file that it stores at a specific location on its filesystem. That notion does not extend well to peer-to-peer systems, because each peer could store the file in a different place. Therefore, content IDs became the best way to identify files on P2P networks. However, content IDs' greatest strength is also their greatest weakness. Content IDs are immutable. A content ID identifies a specific version of a file, because it is identifying a specific version of its contents. Immutability is desirable for many file-sharing use cases, because once a version of a piece of software is released, that version does not change. The same applies to multimedia. Once a picture, movie, or piece of music is published, it does not change. However, the content ID of a version of a software program does not identify the latest version of that software program, because when the contents of that file changes, it will get a completely new content ID,

unrelated to the original one. The functionality to get the latest version is desired though, because users often want the latest version of a piece of software, or of a website, or of some other update-able file. By design, changing the contents of a file changes the content ID of the directory that contains it too, so a mutable file cannot be used inside a directory who's content ID is assumed to be fixed. To provide mutable files, signature addressing can be mixed with location addressing. An identifier for a file can be the signature ID of its publisher prepended onto the path of the file. This informally means: "the file at a location X in a namespace signed by Y". This is the approach that IPFS's mutable identifiers called the InterPlanetary Naming System (IPNS) take [13], as well as Dat's identifiers [17]. Mixing and matching the kinds of identifiers in other ways can be useful too. An immutable file could be identified by its parent directory's content ID prepended to a location ID path. This informally means: "the file at a location X in a namespace who's data corresponds to Y". That kind of ID is useful if the user wants to access many files in the same directory, because they only need to lookup the content ID (which is a large number and not very user friendly) of the directory, and then can access the contained files off of that using familiar location IDs.

Links to files in content-addressed file-sharing systems (systems that use content IDs to identify files) are often just the content IDs themselves encoded for URLs. IPFS uses an encoding called multihashes which include which hash function was used, and how much of the output is used [13] [22]. Therefore, many IPFS IDs start with the base-58 encoded characters "Qm", which refers to using the 256-bit version of SHA-2 called SHA256. Examples of these non-traditional URLs are:

- /ipns/QmSrPmbaUKA3ZodhzPWZnpFgcPMFWF4QsxXbkWfEptTBJd - which is a signature ID of some IPFS data (notice *ipns* not *ipfs*).
- /ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG/readme - which is the location ID of a file in a content-addressed directory on IPFS.
- dat://778f8d955175c92e4ced5e4f5563f69bfec0c86cc6f670352c457943666fe639 - which is a signature ID of some data on Dat.
- https://datbase.org/dat://778f8d955175c92e4ced5e4f5563f69bfec0c86cc6f670352c457943666fe639/contents/dat.json - which is a location ID of a file relative to a signature ID of some data on Dat, which is proxied at a location ID on the traditional web.

- magnet:?xt=urn:btih:c12fe1c06bba254a9dc9f519b335aa7c1367a88a - which is a content ID of some BitTorrent data.

BitTorrent uses .torrent files instead of links to identify the data, but the files contain some metadata, content IDs of the chunks of the file, and URLs of trackers or IP addresses of peers that know where to find that file [12].

2.4 Aberer & Despotovic's Trust Model

The trust model described in "Managing Trust in a Peer-2-Peer Information System" by Aberer and Despotovic uses a social reputation system based on complaints [6]. In this model, peers can be either trustworthy or untrustworthy. Peers default to being trustworthy, until enough other peers complain against them. Complaints are permanent two-way claims that the other peer acted poorly, or that the other peer is lying about this peer acting poorly. That is, given p and q as two interacting peers, if p thinks that q misbehaved (by performing an attack, returning bad results, etc.) then p will submit a complaint about q. Game-theoretically q should claim that p is lying, so q complains about p. Therefore, when one peer complains about the other, the system automatically files a counter-complaint, so complaints are two-way. Given two initially trustworthy peers, and a two-way complaint, an outside observer cannot distinguish who the real adversary is. It is tempting to think that the peer who submitted the complaint first is the honest one, but the adversary could proactively complain against its target. Therefore, in this case, the trust system cannot decide who is untrustworthy. Now, consider that q interacts with another peer r, and r thinks that q misbehaved; r can complain about q (which also means that q complains about r), meaning that q has two complaints about it, but the complainers p and r only have one against each of them. Therefore, it is more likely that q is untrustworthy than both p and r are untrustworthy [6].

Note that the Aberer & Despotovic paper counts two separate complaint variables for complaints filed and complaints against [6]. However, all the calculations using those variables use the product of them. An adversary could then never complain against anyone else and have a complaints filed count of zero. If that were the case, then even with a very high complaints against score, their product would be zero, and the adversary would be trustworthy. Therefore, I have slightly modified the system so that filing a complaint against a peer automatically files a complaint by them about the complainer. This means that the

two complaint variables are always equal, and can be reduced to one count of complaints that a peer is involved in. This model makes sense because it removes the ability to cheat the calculation, and it has already been shown that game-theoretically complaints should be two-way.

The calculation to determine if a peer is trustworthy or not given global knowledge about complaints is presented as a function called *decide* in the paper [6]. It is reproduced in simplified form in Equation 2.1, where the number of complaints a peer has takes the place of the complaints against and complaints filed variables as discussed above. Equation 2.1 calculates peer q's reputation, where $c(q)$ is the number of complaints against q, and $c(avg)$ is the average number of complaints against a peer in the system. A return value of 1 means trustworthy, -1 means untrustworthy.

$$decide(c(q)) = \begin{cases} 1, & \text{if } c(q)^2 \leq \left(\frac{1}{2} + \frac{4}{c(avg)}\right)^2 c(avg)^2 \\ -1 & \text{otherwise} \end{cases} \tag{2.1}$$

Note that the equation presented here is simplified from the one in the paper, because of the differences in the global knowledge store. The variables used in the paper have to be normalized, because the lookups in their global knowledge store are probabilistic, so they do many lookups and normalize the results [6]. In this thesis, the reputation system is tweaked to use the Kademlia DHT that is already being used as a global knowledge store of search results. Since Kad's lookups are not as probabilistic, and return the values that are stored if they exist in the connected network, this normalization step is not needed and has been omitted here.

In the simplest form, a peer p wishing to evaluate the reputation of another peer q needs to lookup the complaint data $c(q)$ from the global data store, and then compute the $decide(c(q))$ function. The reputation paper calls this algorithm $ExploreTrustSimple(p, q)$. However, consider the design of the global knowledge store, which is made up of peers of questionable trustworthiness. If there is room to believe that the peers returning the complaint data might be untrustworthy, their trustworthiness can be explored recursively. The paper calls this algorithm $ExploreTrustComplex(p, q, l)$, where peer p calculates the reputation of another peer q, with a recursive depth limit of l [6]. The algorithm is presented

in Figure 2.4, where W is the set of witnesses who stored information about the complaints, w_i are the witnesses in the set W, and c_i denotes the complaint information from witness w_i.

```
def ExploreTrustComplex(p, q, l):
    if l <= 0:
        return 0
    else:
        W = GetComplaints(q)
        update statistics with W
        if |W| == 0:
            return 0
        if |W| == 1:
            t = ExploreTrustComplex(p, w_1, l-1)
            if t == 1:
                return decide(c_1(q))
            else:
                return 0
        if |W| > 1:
            s = sum(i in 1..|W|, decide(c_i(q)))
            if s > 1:
                return 1
            elif s < 1:
                return -1
            else:
                for i in 1..|W|:
                    if ExploreTrustComplex(p, w_i, l-1) < 1:
                        W = W - {a_i} # eliminate non-trusted witnesses

                s = sum(i in 1..|W|, decide(c_i(q)))
                if s > 0:
                    return 1
                elif s < 0:
                    return -1
                else:
                    return 0
```

Figure 2.4: Modified *ExploreTrustComplex* algorithm.

3. DESIGN

The goal of this thesis was to design *S4h*, a search engine with the following features:

1. **Peer-to-Peer**: so that no central party controls the system.
2. **Secure**: the system should be resistant to censorship, promoting, data mining, and other attacks.
3. **Raw File Search**: the search engine should be able to work without links between files (not just HTML files).

Since the problem of mapping from IDs (e.g. content IDs, signature IDs, URLs, etc.) to files is already well solved by many established systems (e.g. BitTorrent [12], IPFS [13], Dat [17]) this search engine only needs to map from search query strings to a list of IDs.

$$\text{search query string} \rightarrow [\text{ID}] \tag{3.1}$$

Since this mapping needs to be in a global map data structure, and the most popular P2P file-sharing systems including BitTorrent, IPFS, and Dat use a Kademlia-based Distributed Hash Table for their global map data structure, this thesis also chooses to use a Kad DHT. Improving the original Kademlia design with ideas from R/Kademlia and S/Kademlia can increase the efficiency and security of the Kad DHT, but simply using a modern Kad DHT for this mapping does not provide all the desired features. A Kad DHT mapping search query strings to IDs is certainly Peer-to-Peer, and could be used to index raw files, but it is not secure against common attacks on search engines. Though S/Kademlia makes it harder for adversarial nodes to join the network (by making it harder for all nodes to join the network), once malicious nodes do join, they can hijack queries in order to censor data, or submit their own advertisements, or perform other attacks without any serious repercussions. Individual nodes who notice the bad behavior can drop the malicious peer from their routing table, but they have no way of informing other peers about the malicious node. This kind of reputation system could be thought of as one using only direct knowledge, where a node must directly observe malicious activity in order to know that a peer is untrustworthy. The search engine provides the functionality of mapping search queries to a list of IDs, then

the reputation system has the job of ensuring the quality of the returned list of IDs. This problem is related to the ranking problem in traditional search engines.

Search query strings to ID mappings are a many-to-many relation in database terms. That means that a search query string such as "cat pictures" may map to many file IDs, since there are many pictures of cats online. It is also possible that a file ID may be a good search result for many different search query strings. For example, a PDF file of the Kademlia paper may be a good result for searches for "P2P systems", "DHT" "Distributed Hash Tables", "Kad", "Kademlia", "File-sharing system", etc. Therefore, a search query will naturally return a list of many search results. Search ranking algorithms are in charge of ordering the results so that the loosely defined "best" results are at the top of the list. Google's famous search ranking algorithm is called PageRank, and uses links between webpages to determine which pages are most important [24]. Pages that receive a lot of incoming links are more important, and especially pages that receive a lot of incoming links from other important pages are important. Pages that are more important are sorted higher up in the results list. Search ranking algorithms using PageRank work very well for webpages, because they have embedded links to other webpages. However, this thesis aims to design a search engine that works for raw files, not just webpages, and since generic files do not have natural links between them, PageRank can not be the solution to the search ranking problem for this system. Continuing the comparison to webpage search engines, search results are added to the DHT not by the DHT itself as they would be in a centralized search engine, but by nodes, who add mappings whether by manually entering them, or using a crawler of file-sharing services, or by some combination. That is a major difference between a centralized web crawler operated by the owners of the search engine and a decentralized method of upload and ranking. The decentralized model means that users have more control over the ranking of the search results, and each search result has an originator who uploaded the search result and is responsible for its quality. That originator node can be though of as an Original Poster (OP) from the vocabulary of Internet forums.

Consider this subproblem of the search ranking problem: removing bad search results. Just like there is not a clear answer to which search results are the best for a given query, there is not a simple answer to which ones are the worst. However, the system should filter out bad or malicious search results. Intuitively, given an example query "cat pictures", the system should not return advertisements for fruit juice, or as another example a query for "Ubuntu

download" should not return .iso files that contain viruses. This requirement implies using a reputation system that would allow peers who are corrupting the network to be identified as untrustworthy. As discussed in Section 1.3, the reputation system must be a social one, meaning that there ought to be some human element to the reputation system. The reputation system described by Aberer and Despotovic in [6] provides a simple distributed method for a coarse reputation system. Nodes in the network have a reputation, either trustworthy or untrustworthy. They are assumed to be trustworthy by default. Then, other peers can complain about the node, thereby negatively affecting both of their reputations. When a node has a lower reputation than the cutoff, which is determined by the average reputation, the node is untrustworthy. More about the reputation system presented in the paper is in Section 2.4. The paper relies on a globally accessible data structure to store the information about the complaints. It presents its own DHT called P-Grid, which is not as widely adopted as Kademlia-based DHTs. Since this thesis relies so much on the previous research and extensions to Kademlia such as R/Kademlia and S/Kademlia, I chose to modify the reputation system to work on Kad. Therefore, a major component of this thesis was adapting the reputation system to the Kademlia data structure that is already being used to store search mappings.

The original reputation paper stored complaints in a global data structure without first-class replication, where a node stores a complaint by sending it to other nodes that it can contact in charge of the keyspace corresponding to the node ID of the complainee. Then, lookups are sent to multiple peers, to try to find the nodes that were contacted with the storage of the complaint. The complaint data is then normalized because the non-uniform routing means that only a subset of the real number of complaints could be looked up by a given node [6]. In converting this system to use Kademlia as a global data store, it became clear that the normalization would not be needed, since Kad lookups are very highly likely to query all K closest peers to the search key, as proven in the Kad paper [5]. In this thesis' adaptation of the reputation system, a node's reputation is stored at the bit inverse of its node ID, replicated across the K closest peers as normal. Normal Kad lookups query those K closest peers to the search key, and reputation lookups do the same. Storing a complaint by node p against node q happens by node p sending its complaint to all K peers closest to $\sim q$, the bitwise inverse of q's node ID. Then, the peers that receive the complaint forward the signed complaint to the K closest peers to $\sim p$, the bitwise inverse of p's node ID. This

method ensures that queries about p or q find the complaint, because p game-theoretically would want to complain against q without having q reflexively complaint about p, but since p complaining to the peers at $\sim q$ automatically triggers that reflexive complaint, there is no way for p to avoid it. Also, since all K peers at $\sim q$ forward the complaint to the peers at $\sim p$, the responsibility of forwarding is not on one peer at $\sim q$, which may be adversarial and not forward the complaint. During lookup, complaints returned by a majority of the K closest peers are considered during calculation of the reputation. If all complaints returned by the K peers were considered, then an adversarial peer could return many fake complaints and tarnish the reputation of a node near its key inverse. Complaints, like all messages in the network, should be signed by the originator and validated by the receiver, so that counterfeit complaints are not stored in the system. Complaints should be refreshed in the same way that search data is, so that complaint data is maintained through churn.

Naively, a node performing a search query could receive a list of search results, each signed by the originator of that data, and then lookup the reputation of each of the originators, in order to eliminate bad results coming from untrustworthy originators. However, each reputation lookup takes $O(log\ N)$ sequential messages, which can introduce a long delay to the whole search query. Instead, nodes should only store new data (including republished data) when the originator is reputable. Therefore, when a node receives a store message, it should lookup the reputation of the originator. Since latency is much less important for store messages than search queries, this method will reduce the cost of a lookup to just the $O(log\ N)$ search query compared to the naive lookup of reputations. The cost of this modification is a level of indirection, where now the peers returning the search results must be trustworthy, so that they do not store malicious data. Therefore, when two nodes p and q are connecting for the first time, and node p is about to add node q to its routing table, node p should lookup the reputation of node q, in order to make an effort to only have trustworthy peers in its routing table. However, querying for a new search term will likely put the searcher in contact with new unknown peers, but to defer the delay of looking up the new peers' reputation, the reputation lookup can wait until after the search query is finished. This trades some security for some speed, but assuming that the peer who referred the searcher to the questionable peer is trustworthy, and that referring peer maintains a routing table of only trustworthy peers, then the lookup is still secure. In order to maintain a routing table of only trustworthy peers, given that peers may be added with a good

reputation and then later on become untrustworthy, all nodes should periodically query the reputation of the peers in their routing tables. One of the benefits of Kademlia over earlier DHTs listed in the Kad paper is that it does not need periodic routing table maintenance [5]. Since Kad uses XOR as a distance metric, and XOR is symmetric, Kad routing tables can be updated when they receive messages. Therefore, the original Kad paper does not describe any messages to specifically update the routing tables, as it happens passively. The S/Kad paper recognized that in order to better handle churn, nodes should have very complete knowledge about their siblings - the K peers closest in the keyspace. Therefore, S/Kad extends the Kad protocol with some messages to maintain information about the siblings [15]. The sibling maintenance messages are efficient though, just between closely connected nodes. Reputation maintenance messages however do full $O(log\ N)$ lookups, which adds a lot of reoccurring maintenance messages to the system (see Chapter 6).

The reputation scores of a node go up as the node is involved in more complaints, as shown in Equation 2.1. Therefore, a node's reputation is decided by its number of complaints relative to the average number of complaints per node. Although this design has a Kademlia global data store, it is infeasible to store all complaints at a single key, because the nodes near that key would get overloaded with messages. To solve this problem, the average number of complaints per node is stored as statistics at each node. Nodes that receive complaints store them and update their statistics. Since node IDs are generated randomly, but not uniformly, there will be some variation in the number of peers that a node is in charge of storing the complaints for. Also, nodes in charge of storing complaints for a peer involved in many complaints will have a higher average number of complaints per node. To compensate for this, complaining nodes could send their complaints to c peers randomly selected from their routing table. This has the effect that well-connected nodes will have a higher expected value for average number of complaints per node, meaning that well connected nodes will have a higher tolerance for complaints before deciding that a peer is untrustworthy. This asymmetric distribution of average number of complaints is not exactly the method described in the reputation paper, but still has the same effect.

Like complaint data, which is replicated across the K closest peers to the bitwise inverse of the subject node, search data is replicated across the K closest peers to the hash of the search query. Search data expires and is republished as described in Section 2.2.1. In this way, data is maintained through churn, and can exist after the original publisher goes offline.

Note that search results correspond to exact string matches of the search query, since they use a cryptographic hash function. In order to improve the user experience, some pre-processing of search queries should be done, to break up the search into multiple queries of the keywords or key phrases as applicable. Search mappings should also be inserted into the DHT at many applicable search query keys, by performing a pre-processing step to associate keywords and keyphrases with the files. Note that automated or semi-automated tools could assist with this process especially for text and to a lesser extent image files.

It may seem like the transport layer for *S4h* could be unencrypted, since the DHT stores globally accessible information. However, encryption of the connections between the peers is important so that eavesdroppers cannot see what a node is searching for. Using a hash function as a one-way function to hash the search query is already being performed, so the plaintext search query is not being sent over the wire. However, a dictionary attack could be used to reverse the hash function and collect data about the search query. Therefore, proper encryption must be used between nodes, such as TLS. Besides encryption, the requirements for the transport layer are minimal. Reliable transport is not needed, because retries would be undesirable since low latency is important for a search engine. The system model is also tolerant to node failure, because nodes leaving the network is a common behavior. The system is also relatively resistant to network partitions, because node IDs are not correlated to geographic location (see Section 2.2.1). Also, as *S4h* is designed for real-world applications, the system assumes asynchronous messaging, where there is no known upper bound on message delay. That being said, search queries will have short timeouts (relative to the measured RTT), so that a slow node does not block a query. Many options for a network protocol that provide encryption and authentication exist. HTTP3 is designed to be used with TLS 1.3 for encryption and authentication, and supports 0-RTT connection setup between previously connected hosts.

To summarize the design of *S4h*: a Kademlia-based DHT provides routing functionality for a peer-to-peer global data store, which is secured by updates to the Kademlia design including R/Kademlia and S/Kademlia, and is used to store mappings from hashes of search query strings to links to files in file-sharing networks. In order to maintain the security of the system and remove malicious peers and search results, a complaint-based reputation system by Aberer & Despotovic is simplified and modified to use the Kad system as its global data store. The peers communicate over authenticated and encrypted connections, and maintain

the search and complaint data throughout nodes joining the network, leaving the network without warning (encompassing crash-failures), and network partitions.

4. RESULTS

Though the design decisions presented in Chapter 3 were motivated by logical arguments, it was important to run some experiments on them. Often when working on software important details show up in implementation that were not considered in the design phase. For this reason, and to test design ideas as they came up, an implementation was developed. It is now stored online at https://github.com/cowang4/S4h, which contains source code and integration tests. The implementation does not completely cover all of the ideas in this thesis, instead it focuses on the main design problem that arose: merging the Aberer & Despotovic reputation system with a Kademlia DHT. Therefore, the implementation provides a simple Kademlia DHT without S/Kademlia or R/Kademlia improvements, and a modified version of the reputation system to work with the Kad implementation. The code is written in the Rust programming language, though in hindsight that was a bad choice. Rust has many benefits, but at the time of writing the asynchronous programming features are not fully developed, and therefore writing an asynchronous networking application in Rust was challenging and diverted attention from more important research work.

The first experiment was to verify that the implementation exhibits the behavior described in the reputation paper, by seeing if the number of complaints needed for a node to become untrustworthy in the implementation matched the expected number of complaints for a node to become untrustworthy from the decide formula given in Equation 2.1. For this experiment, the value of all peer's average number of complaints per node was held as the independent variable, which provides the benefit that the test is not dependent on the size of the simulated network. The reputation score threshold is the maximum reputation score that a trustworthy peer can have, any higher and the peer is untrustworthy. The reputation score threshold from the decide formula is given in Equation 4.1 where x is the average number of complaints per peer, and is graphed as the black line in Figure 4.1.

$$\text{reputation\_score\_threshold}(x) = \left(\frac{1}{2} + \frac{4}{x}\right)^2 x^2 \tag{4.1}$$

The hypothesis of this first experiment was that the reputation score corresponding to the minimum number of complaints required for a peer to become untrustworthy in the simulation is greater than the reputation threshold from Equation 4.1. The reputation

score for the peer in the simulation is calculated by Equation 4.2 where x is the number of complaints that the peer has been involved in.

$$\text{reputation\_score}(x) = x^2 \tag{4.2}$$

The simulation used 20 nodes with random node IDs, named here by order of creation, with a key size of 32 bits, a K of 6, an α of 3, and each peer was bootstrapped to the first two peers. Then, each peer looked up its own node ID to populate its routing table. Next, the 7th peer (chosen arbitrarily from the non-bootstrap nodes without loss of generality) submitted a search result. In this simulation, each peer looked up the search result, decided that it is bad enough to warrant a complaint, complained about peer 7, and then queried the reputation of peer 7. By the nature of the system, peer 7 started out trustworthy, may have remained trustworthy for the first few complaints (depending on the reputation score threshold), then became untrustworthy after a number of complaints. The number of complaints needed for peer 7 to become untrustworthy is the dependent variable that is being measured in this experiment. The results are shown in Table 4.1 and are graphed in red in Figure 4.1.

Table 4.1: The results of the experiment to verify the number of complaints until trustworthiness.

avg complaints per peer	complaints to untrustworthy	reputation score
0	1	1
0.0001	5	25
0.001	5	25
0.01	5	25
0.1	5	25
0.2	5	25
0.3	5	25
0.4	5	25
0.9	5	25
1	5	25
2	6	36
3	6	36
4	7	49
5	7	49

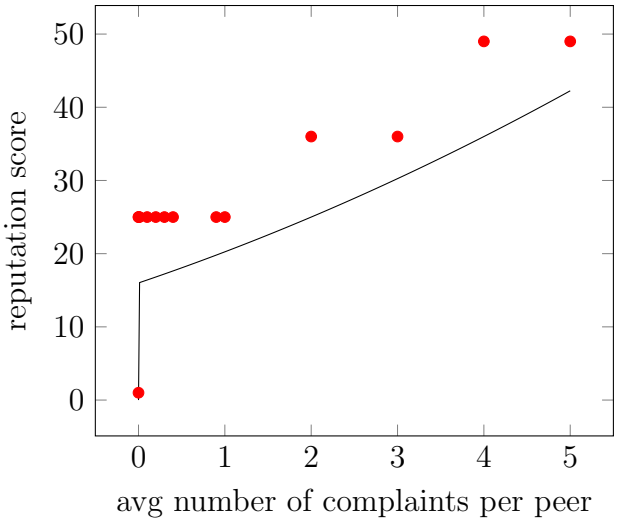

Figure 4.1: The expected reputation score threshold graphed in black, based on the average number of complaints per peer x, and the experimentally measured number of complaints to untrustworthy in red.

The previous experiment held the average number of complaints per node as the independent variable, but the value of that variable strongly dictates the effectiveness of the reputation system, as described in Chapter 3. The reputation paper leaves the exact calculation of the average number of complaints for future work, so this second experiment tests a possible implementation. The design is for each node to locally store the average number of complaints per node so far, and a count of how many reputation queries that it has done. Then, every time a node queries a peer's reputation, it updates the average by Equation 4.3, where $avg\_num\_complaints_i$ is the average number of complaints per node after i queries and $num\_complaints$ is the number of complaints that the currently looked up peer is involved in. The count of reputation queries i is incremented by one as well.

$$avg\_num\_complaints_{i+1} = \frac{(i * avg\_num\_complaints_i) + num\_complaints}{i+1} \tag{4.3}$$

To test that the locally-computed average number of complaints per node increases as more complaints are filed in the network, a simulation-based experiment was run. The simulation setup was similar to the first experiment, comprised of 20 nodes with random node IDs, named here by order of creation, with a key size of 32 bits, a K of 6, an α of 3,

each peer was bootstrapped to the first two peers, and looked up its own node ID to fill in its routing table. The average number of complaints per node is initialized to 0.01, and the i is initialized to 1. In this experiment, the number of ambient complaints against each peer in the network is the independent variable, such that after the network is setup, x peers complain about all other peers, meaning that all peers have at least x complaints against them. Then, the number of complaints until untrustworthiness was measured after zero, ten, and twenty reputation lookups. This means that the node complaining about peer 7 first does zero, ten, or twenty lookups of the other peers' reputation, then complains about peer 7, and finally queries peer 7's reputation. The number of complaints until untrustworthiness measured at zero is the control, and is just the initial value averaged with the number of complaints against peer 7, because the reputation paper specifies updating the statistics before deciding the trustworthiness, as seen in Figure 2.4. The hypothesis of this experiment is that the number of complaints until untrustworthiness measured at ten and twenty should increase as the number of ambient complaints in the network increases. The results of the experiment are presented in Table 4.2, the results for zero reputation queries are graphed in Figure 4.2, the results for ten reputation queries in Figure 4.3, and the results for twenty reputation queries in Figure 4.4.

Table 4.2: The results of the experiment using the computed average number of complaints per node.

ambient complaints per node	complaints to untrustworthy using 0 reputation queries	complaints to untrustworthy using 10 reputation queries	complaints to untrustworthy using 20 reputation queries
0	6	5	5
1	6	5	6
2	6	6	6
3	6	6	7
4	6	6	8
5	6	7	9

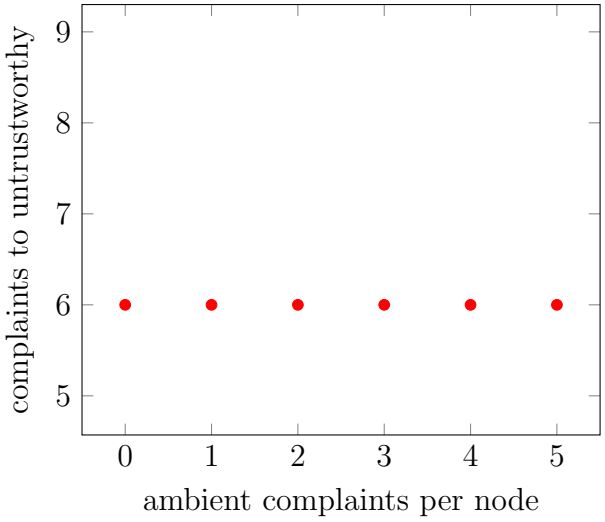

Figure 4.2: The number of complaints until untrustworthy vs. the number of ambient complaints per node using 0 prior reputation queries.

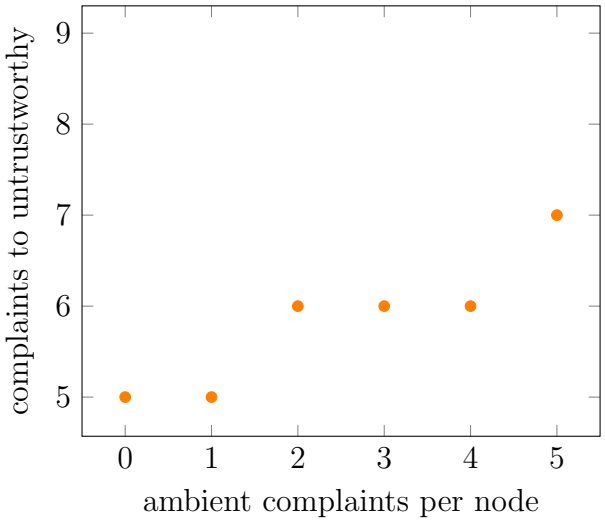

Figure 4.3: The number of complaints until untrustworthy vs. the number of ambient complaints per node using 10 prior reputation queries.

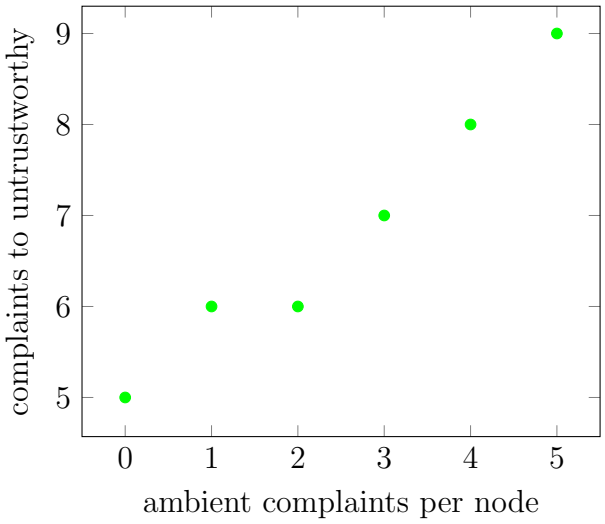

Figure 4.4: The number of complaints until untrustworthy vs. the number of ambient complaints per node using 20 prior reputation queries.

5. DISCUSSION

The implementation experiments proved to be very insightful. The reputation system as described in the paper separates the variables for the number of complaints against or received by a node and the number of complaints filed by a node [6]. Also, the paper suggests that to submit a complaint, the complaining node p should insert the complaint about the complained about node q at the key corresponding to q (in this design $\sim q$), and a record of the complaint at the key corresponding to its own node ID (in this design $\sim p$). However, game-theoretically p will not submit a record of the complaint at its own corresponding key $\sim p$, because that will hurt p's reputation, by increasing the number of complaints that p has filed. A simple fix for this problem is for the peers at $\sim q$ to submit the record that p has submitted a complaint to $\sim p$ when they receive the complaint from p. Then, when q learns about p's complaint - presumably by looking up its own reputation - it will game-theoretically complain back, by submitting a complaint to $\sim p$, which will trigger a record stored at $\sim q$. Note that all calculations involving the number of complaints filed and received combine the two values by multiplying them. As seen in the version of the *decide* function presented in the paper [6]:

$$decide(cr(q), cf(q)) = \begin{cases} 1 & \text{if } cr(q)cf(q) \leq \left(\frac{1}{2} + \frac{4}{\sqrt{cr(q)cf(q)}}\right)^2 cr(q)cf(q) \\ -1 & \text{otherwise} \end{cases} \tag{5.1}$$

Given that the values of complaints filed and received are always used together and that game-theoretically q will always counter-complain about p, an optimization presented in this thesis is to combine the two variables into just a count of the number of complaints that a node is involved in, which simplifies the decide function in Equation 5.1 to the Equation 2.1, by replacing $cr(q)cf(q)$ with $c(q)^2$. The other optimization from this implementation work is that the peers at $\sim q$ should automatically forward the complaint to $\sim p$, which has the combined effect of both registering the complaint filed and submitting a corresponding counter-complaint. This fix also has the benefit of eliminating a reputation attack where an adversarial node could never file a complaint about any other peers, meaning that its number

of complaints filed is zero and therefore its reputation score will be zero, which will always be below the average and therefore always be reputable. No matter how many complaints that the node has received, the reputation calculation in the original *decide* multiplies the number of complaints filed with the number of complaints received, so if the number of filed complaints is zero, a node could always have a perfect reputation.

5.1 Discussion of Experiments

The first experiment's hypothesis was that the number of complaints until peer 7 became untrustworthy would be greater than or equal to the expected reputation threshold from the decide function. The results in Table 4.1 agree with the hypothesis. The experimental results are the number of complaints until peer 7 is untrustworthy, which is an integer count, when squared is the reputation score of peer 7. The reputation scores are graphed in 4.1 as the red points, which lie at the minimum integer squares above the expected reputation threshold graphed as the black line.

The second experiment's hypothesis was that the reputation score threshold would increase as the number of ambient complaints in the network increased. In this experiment the control test sampled the number of complaints to untrustworthy using zero prior reputation queries, meaning that the locally-stored averages were still at their defaults. It appears that when averaged with the default average number of complaints, the threshold is high enough that six complaints is the minimum number required to be untrustworthy in the control case. The control test does not depend on the number of ambient complaints, so the results are constantly six. In the samples that used ten prior reputation queries, we see that the threshold increases some as the number of ambient complaints increases, which agrees with the hypothesis. Though, using more prior reputation queries increases the reputation threshold faster to the expected, as seen in the samples using twenty prior reputation queries. The results agree with the hypothesis and motivates the use of this kind of method to maintain the critical average number of complaints per peer statistic.

5.2 Implementation Summary

Overall, the task of adapting the reputation system to use a Kademlia data store went smoothly, owing to the modular way that the reputation system was designed and the flexibility of the DHT to store arbitrary data. Another insight from the implementation is

that whether a node is trustworthy or not is highly dependent on the average number of complaints that nodes in the network are involved in. A network with more complaints per node on average will tolerate a higher number of complaints against a node before deeming it untrustworthy.

The successful port of the reputation system to work in the Kad network means that *S4h* provides the ability to complain about a peer when the peer is observed performing some attack. It is possible to complain when the peer is the originator of a irrelevant search result, or when the peer forges messages from other peers by checking the authentication of all messages. When a node discovers that its peer is untrustworthy, it can drop the peer from its routing table, and put the peer on a blacklist. Once removed from all the routing tables, the adversary will not receive any messages from any peer in the network. This functionality means that malicious nodes (as long as their malicious action is detected) will eventually be removed from the network. This feature provides resistance to censorship and promoting attacks compared to a P2P search engine without an explicit trust system.

6. FUTURE WORK

While this thesis presents a design for a search engine with explicit trust to solve the problem of removing malicious peers from the network, it does not present solutions to many other issues that crop up in real-world applications of this nature. For instance, the search result ranking problem for raw files presents an interesting mix of challenges in the distributed networks, search, and user experience fields. One potential solution is modeled after ranking systems in online recommendation sites, such as Amazon reviews and Reddit upvotes. The Reddit model translates nicely to search ranking, where each peer has a reputation score (karma) based on the number of up and down votes that their published results get. Of course, that solution presents challenges, such as how to combat fake upvotes. The opportunity to support mobile peers that are only online briefly presents some challenges too; perhaps a they could rely on a trusted permanent peer, such as in DNS. The Kademlia DHT and reputation system are probabilistically modeled in their own papers, but there is an opportunity to formally specify and prove how these systems work and interact with each other. Research based on real-world experience with Kademlia DHTs in BitTorrent and other deployed networks will certainly provide insight for optimizations and opportunities to harden the design. More advanced DHTs have been developed too, the Coral Distributed Sloppy Hash Table (DSHT) is used in IPFS and brings some interesting ideas to the field [23]. New cryptographic puzzles could potentially suit the application of securing node ID generation better than proof-of-work, including proof-of-stake, or proof-of-space methods. Locality-sensitive hash functions could be used on search queries to group similar queries together, but the performance and load-balancing implications of that choice need to be explored. All of that being said, the market for a well-implemented secure P2P search engine is wide open. Another detail that is open to more future work is the exact method for maintaining the complaint statistics. This thesis experimented with a simple average, though a exponentially weighted moving average (EWMA) may provide a more accurate measure for long-running nodes.

7. CONCLUSION

In conclusion, this thesis presents a design for a P2P search engine that has first-class trust using a complaint-based reputation system. I ran experiments on an implementation of the design and found that the implementation of the combined Kademlia and reputation system could judge a peer's reputation according to the specified given fixed statistics and could derive reasonable statistics using standard reputation queries. This shows that *S4h*, search engine design presented here, can hold peers accountable for their interactions and remove untrustworthy ones from the network, thereby hardening the system against common attacks such as censorship and promoting. The main contribution of this work is the novel combination of a reputation system and a Kademlia-based DHT to build a P2P search engine, and the modifications of the two parts necessary to fuse them together securely.

LITERATURE CITED

[1] "FORTUNE 500: 1972 Archive Full List 1-100", Archive.fortune.com. [Online]. Available: https://archive.fortune.com/magazines/fortune/fortune500_archive/full/1972/. [Accessed: 07- Mar- 2019].

[2] J. Monegro and C. Burniske, Placeholder Thesis Summary. 2017, pp. 1-7. Available: https://ipfs.io/ipfs/QmZL4eT1gxnE168Pmw3KyejW6fUfMNzMgeKMgcWJUfYGRj/PlaceholderThesisSummary.pdf. [Accessed 07-Mar-2019].

[3] "Decentralization", Mastadon Documentation, 2018. [Online]. Available: https://docs.joinmastodon.org/usage/decentralization/. [Accessed: 07- Mar- 2019].

[4] "Overlay Network, VXLAN", Ssup2 Blog, 2019. [Online]. Available: https://ssup2.github.io/images/theory_analysis/Overlay_Network_VXLAN/Overlay.PNG. [Accessed: 07- Mar- 2019].

[5] P. Mymounkov and D. David Mazires, "Kademlia: A Peer-to-peer Information System Based on the XOR Metric", 2002. Available: https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf. [Accessed 7 March 2019].

[6] K. Aberer and Z. Despotovic, "Managing Trust in Peer-2-Peer Information System", in CIKM, Atlanta, GA, USA, 2001, pp. 310-317.

[7] "DuckDuckGo Privacy", DuckDuckGo, 2019. [Online]. Available: https://duckduckgo.com/privacy. [Accessed: 07- Mar- 2019].

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, "A Scalable Content-Addressable Network", in SIGCOMM, San Diego, CA, USA, 2001, pp. 161-172.

[9] H. Balakrishnan, M. Kaashoek, D. Karger, R. Morris and I. Stoica, "Looking Up Data in P2P Systems", Communications of the ACM, no. 462, pp. 43-48, 2003.

[10] B. Y. Zhao, Ling Huang, J. Stribling, S. C. Rhea, A. D. Joseph and J. D. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," in IEEE Journal on Selected Areas in Communications, vol. 22, no. 1, pp. 41-53, Jan. 2004.

[11] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems", in Middleware, Heidelberg, Germany, 2001, pp. 329-350.

[12] B. Cohen, "The BitTorrent Protocol Specification", BitTorrent.org, 2001. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html. [Accessed: 07- Mar- 2019].

[13] J. Benet, "IPFS - Contend Addressed, Versioned, P2P File System", GitHub, 2014. [Online]. Available: https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf. [Accessed: 07- Mar- 2019].

[14] B. Heep, "R/Kademlia: Recursive and topology-aware overlay routing," 2010 Australasian Telecommunication Networks and Applications Conference, Auckland, 2010, pp. 102-107.

[15] I. Baumgart and S. Mies, "S/Kademlia: A practicable approach towards secure key-based routing," International Conference on Parallel and Distributed Systems, Hsinchu, 2007, pp. 1-8.

[16] Douceur, John R (2002). "The Sybil Attack". Peer-to-Peer Systems. Lecture Notes in Computer Science. 2429. pp. 25160. doi:10.1007/3-540-45748-8_24. ISBN 978-3-540-44179-3.

[17] M. Ogden, K. McKelvey and M. Madson, "Dat - Distributed Dataset Synchronization and Versioning", datproject.org, 2017. [Online]. Available: https://github.com/datprotocol/whitepaper/raw/master/dat-paper.pdf. [Accessed: 07- Mar- 2019].

[18] "Tribler: an attack-resilient micro-economy for media", GitHub, 2019. [Online]. Available: https://github.com/Tribler/tribler/wiki. [Accessed: 07- Mar- 2019].

[19] U.S. Department of Commerce, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", NIST, Gaithersburg, MD, USA, 2015.

[20] M. Saarinen and J. Aumasson, "RFC 7693 - The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)", RFCs, 2015. [Online]. Available: https://tools.ietf.org/html/rfc7693. [Accessed: 07- Mar- 2019].

[21] J. Oliver, C. Cheng and Y. Chen, "TLSH - A Locality Sensitive Hash", in 4th Cybercrime and Trustworthy Computing Workshop, Sydney, Australia, 2013.

[22] "Multihash", Multiformats.io, 2019. [Online]. Available: https://multiformats.io/multihash/. [Accessed: 07- Mar- 2019].

[23] M. Freedman and D. Mazieres, "Sloppy Hashing and Self-Organizing Clusters", Lecture Notes in Computer Science, vol. 2735, 2003. Available: https://www.researchgate.net/publication/2901175_Sloppy_Hashing_and_Self-Organizing_Clusters. [Accessed 7 March 2019].

[24] L. Page and S. Brin, "The PageRank Citation Ranking: Bringing Order to the Web", 1998. [Online]. Available: http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf. [Accessed: 07- Mar- 2019].

[25] "Application Usage & Threat Report", Research Center, 2013. [Online]. Available: https://researchcenter.paloaltonetworks.com/app-usage-risk-report-visualization/. [Accessed: 07- Mar- 2019].

[26] T. Seppala, "The Pirate Bay shutdown: the whole story (so far)", Engadget, 2014. [Online]. Available: https://www.engadget.com/2014/12/16/pirate-bay-shutdown-explainer/. [Accessed: 07- Mar- 2019].